MC/DC - TNT:

Monte Carlo / Dynamic Code - Toy Neutronics Testbed [9]

An event based transient Monte Carlo testbed for investigations into parallelization techniques in Python, based on methods from MC/DC

Jackson P. Morgan

Center for Exascale Monte-Carlo Transport (CEMeNT), Oregon State University

March 17th 2022

Contents

1 Introduction

Monte Carlo / Dynamic Code (MC/DC) is a soon-to-be released Python-based neutron transport solver, developed as part of CEMeNT (Center for Exascale Monte Carlo Neutron Transport). It is a research code used to investigate novel methods for the development of dynamic simulations. As Monte Carlo (MC) neutron transport applications are often highly computationally taxing, a Python-based technique to rapidly gain parallelism for both GPU and CPU hardware is required so methods implemented in MC/DC can be tested at the high performance computing (HPC) scale. Our objective is to find a technique that presents the best software engineering solution to enable rapid prototyping for methods research in MC/DC at the HPC scale using both CPUs and GPUs.

We implemented core components of MC/DC in a testbed to allow for even faster exploration of parallelism in a Python framework. This testbed, MC/DC-Toy Neutronics Testbed (MC/DC-TNT), is a transient, event-based, mono-energetic solver that enables straightforward parallelization of its compute functions. In this work we explore CPU based acceleration though we examine GPU features of both methods considered.

2 Methodology

As this test-bed's primary goal is software engineering methods, our physics implementation is relatively simple, where test problems have many well defined reference solutions. This is done on propose to enable rapid implementation of the acceleration methods presented later. We target many features of (MC) neutron transport for implementation on this bench that are well defined by literature.

Event based transport

Event based MC methods are commonly implemented to decrease the divergence of a code for better optimized performance on a GPU. This is done by taking all particles through a given event at the same time (lots of for-loops over the total number of particles alive). Event-based methods take all particles undergoing a given event through the same series of operations with similar data, thus produced functions are vary readily parallelizable in general, and especially for vector based devices.

Slab geometry

Implementing new methods and simulation packages in so called "slab geometry" (aka plane wall geometry) is standard in non-production codes, as it allows method and computational exploration without requiring implementation of complex geometry functions. Particles will still undergo three dimensional transport within the slab, but changes in y and z are unimportant as it is assumed that materials are homogeneous in that coordinate plane.

Surface tracking

Surface Tracking is a method of handling multiple regions where the minimum between the nearest region boundary

$$||D_{\text{bound}}|| = \frac{i * dx - x}{\mu} + \text{kicker} \tag{1}$$

and a sampled distance to collision

$$||D_{\text{collision}}|| = \frac{-ln(\xi)}{\Sigma_{\text{total, i}}} \tag{2}$$

is used to find the next location of the particle; where x is the current location of the particle, i is the known mesh cell, dx is the uniform mesh cell width, the kicker $= 1\times10^{-10}$, ξ is a psudeo-random number between [0,1], μ is the $\hat{\imath}$ unit vector, and $\Sigma_{\text{Total, i}}$ is the total macroscopic cross-section in the i^{th} mesh cell [6]. If the boundary is closer the particles is moved to the boundary (plus a small kicker to avoid computation issues) and a new distance to collision is found. Normally this is done only at interfaces of differing material, in TNT this is done between each mesh boundary to avoid divergence. We expect the extra overhead of more re-computation of distances to be much smaller than the alternative.

Also when implementing Surface Tracking a scalar flux can be computed. In the transport kernel that implements surface tracking, a cell wise accumulator of the total distance traveled by all neutrons in a cell (D_i) as well as that distance squared (D_i^2) (for an associated error estimate) is tallied and stored. At the end of the simulation average cell-wise scalar flux ($\bar{\phi_i}$) can be found by

$$\bar{\phi_i} = \frac{D_i}{Ndx} \tag{3}$$

where D_i is the total distance accumulated in the i^{th} cell, N is the total number of initially supplied particles, and dx is the cell width [8]. We can also compute an associated cell wise error by

$$\sigma^2 = \frac{\overline{D_i^2} - (\overline{D_i})^2}{N-1} \tag{4}$$

Live fission particle production

Live fission particle production is something that a lot of MC neutron transport codes do not do, and is inherit to the transient method which MC/DC is built upon. When a fission occurs it's prompt (instantaneously generated) neutrons are produced at the same point in time and space where the fission happened. These daughter particles can then go on to have more interactions through out the problem. This means that the population of the particles undergoing transport is not constant and is changing as the simulation is cycled through.

Scattering, fissioning, and sourcing isotropically

Directions are sourced isotropically for an initial allocation of particles, after a scattering event, or after fission event. First two random angles for spherical coordinates are found, a zenith

$$\mu = 2\xi - 1$$

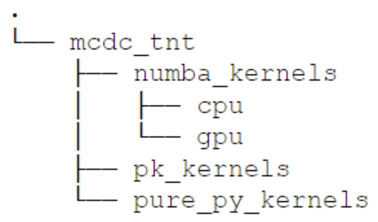
and an azimuth

$$\theta = 2\pi\xi$$

where ξ is a pseudo-random number between 0 and 1 (new for each operation) [8]. These angels are then converted to Cartesian unit vectors via

$$c = \sqrt{1-\mu^2}$$
$$\hat{\imath} = \mu$$
$$\hat{\jmath} = \cos(\theta) * c$$
$$\hat{k} = \sin(\theta) * c$$

and stored for every particle.



```
.
└── mcdc_tnt
    ├── numba_kernels
    │   ├── cpu
    │   └── gpu
    ├── pk_kernels
    └── pure_py_kernels
```

Figure 1: Layout of the package

3 Implementation

3.1 Structure and Layout

MC/DC-TNT is designed with modularity at its core. All that is required to alter a acceleration implementation is the alteration of the import command in the function that runs the kernel. As such, sub-modules are set up around which acceleration scheme they seek to implement. Currently pure Python, Numba, PyOmp, and Pykokkos kernels are implemented as importable sub modules. Figure 1 shows the directory and thus module layout.

All simulations start with the `run` function which handles command line argument parsing, imports from a `yaml` input file, and calls `generations`.

Figure 2 shows the layout of the `generations` function, where the bulk of the computation takes place. First the phase space is allocated in vectors that are sized larger than the initial allocation of particles, to accommodate fission particles, where the index refers to the particle value. For example: the i^{th} location of the x-position and x-direction vectors refer to those elements of the same particle. These vectors are then filled with an initial allocation through the `SourceParticles` function. Then while any particle is still alive, `generations` will loop over all transport functions until all particles are dead. Each function does a specific operation related to event based transport:

1. `SourceParticles` sources particles at the beginning of a simulation within a generating region. Sets directions isotropically
2. `Advance` implements the surface tracking algorithm to move particles to their next event location. Also tallies track lengths for use in computation of a scalar flux and associated error. This is the most computationally intense function, taking well over 90% of run time.
3. `StillIn` determines if particles are still in the problem space. If they have exited the function sets them to dead and tallies the event
4. `SampleEvent` determines particles event based off material data. If captured sets to dead.
5. `Scatter` samples a new direction
6. `FissionAdd` produces daughter particles at same location in space and time with new random direction. Sets parent particle to dead.
7. `BringOutYourDead` cleans phase space vectors of dead particles.

All transport functions are implemented with acceleration schemes to speed up the computation.

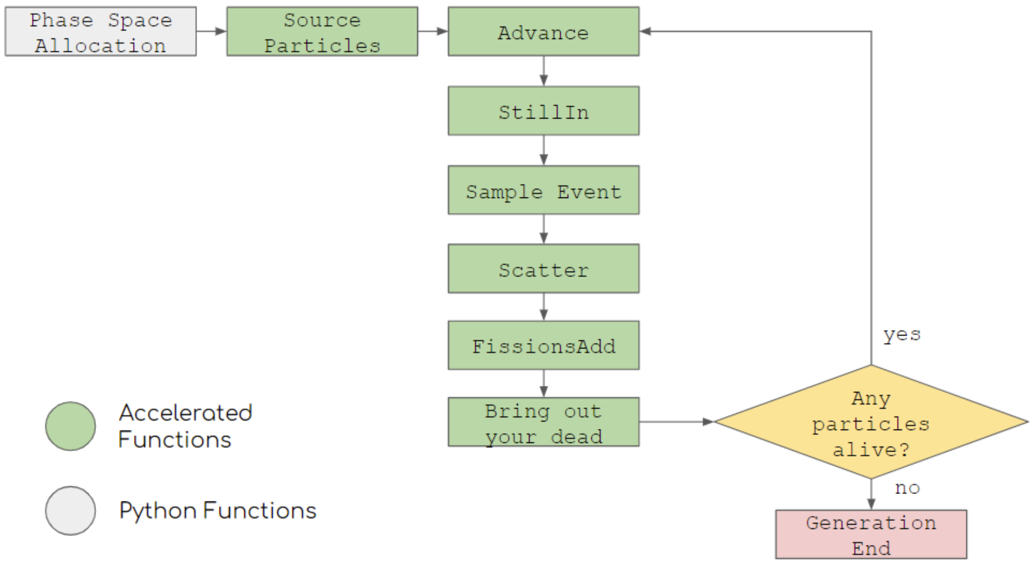


Figure 2: Flow chart of `generations` function showing which sub functions are accelerated.

3.2 Paralleization Techniques

We examine two methods of parallelization within a Python framework. Both operate by using Python as "glue" language (a language which is used to run other compiled kernels) with a just-in-time (JIT) compilation scheme to produce and run compute kernels. Both are also coupled with automatic code translation protocols, to aid in the rapid development of these kernels, with the source being written in full Python.

While this work does not currently implement GPU functionality, we explore both methods' GPU abilities.

3.2.1 PyKokkos

PyKokkos [1] is a Python library implementation for the `C++` Kokkos HPC portability model developed by Sandia National Laboratories [2]. PyKokkos is under active development and does not implement all the functionality of Kokkos. PyKokkos currently only implements CUDA and OpenMP back-ends.

PyKokkos has the ability to abstract away the production of compute kernels from their target hardware. It even has the ability to abstract data structures from the user, appearing almost as type-annotated NumPy [4] arrays. While the programming model is entirely Python-based (the user only programs in Python) it can be non-intuitive, as it is based on a paradigm specifically designed for abstraction.

Note that PyKokkos's instillation is difficult and as such it is not expected that a casual user will install it. Thus it's testing functions have been disabled for this specific work. However instructions that seem to work for Unix based systems are included in the `README`, though mileage may vary.

3.2.2 Numba + CUDA

Numba [5] is a translator and compiler for Python that implements the LLVM high performance compiler library, and promises "C-like speeds" for functions that implement it. In simple cases, nothing more than a compiler flag is required to produce a C kernel for a given function that is automatically bound and run by Python. This can even apply to simple functions to be parallelized on a CPU, where parallelization can be implemented with one flag option above a function. However, when functions become more complex, the use of more Numba commands in the Python code is required. It directly supports Nvidia GPUs and seems to be the method of choice for interfacing with the CUDA API from Python.

Recently a sub-module to Numba has been developed called `pyomp` [7] that seeks to allow Numba kernels to take advantage of the OpenMP API for CPU threading. We explore this sub-module and Numba's native "threading" techniques separately from one another as they require slightly different implementations.

While Numba does have some abstractions for specific GPU operations, most of these cannot be leveraged to offload the computational burden in our simulations. In our use case the user will effectively have to write CUDA C kernels within Python. Thus, the user will have to be aware of the hardware architecture, hampering rapid prototyping. Also, Oden found that Numba does include a significant computational overhead, reaching 50–85% of the performance of pure CUDA C versions of compute-intensive benchmarks [10]. In spite of this, we include it in this comparison due to its increasing popularity, ease of use for CPU implementations, and industry support.

3.3 Interactions

There are two ways to interface with MC/DC-TNT. Through the Python interpreter or using `mcdc_tnt.run(input.yaml, output.out, hardware target)` (after importing), where the latter two inputs are not required. Or through the command line interface for the `run.py` file in the `mcdc_tnt/` directory, where the user must declare the input file, and can, if desired, specify an output file and acceleration scheme. In both cases the acceleration scheme is also described in the `input.yaml` file, through a command line request/interpreter input will supersede it. As the nature of the inputs are relatively complex the input file is required. For examples of running either case, and documentation about the input file, see the `README`.

A full integration test is available in the `test` directory along unit tests and a stable reference solution for validation. After the environment has been set up and `mcdc_tnt` has been installed (both using commands found in the `README`) the integration test can be performed with `python test_hardware.py`. Note that due to PyKokkos's difficult instillation, it's tests have been disabled, as it is assumed the user will not be installing it. Also because PyOmp is a conflicting package to Numba it is also disabled for this test.

3.4 Dependencies

Dependencies are listed verbosely in the `requrments.txt` file within the `mcdc_tnt` package. The main requirements are:

- Numpy - to handle data vectors and conduct speedy operations in Python
- Numba - acceleration scheme
- Numba: PyOmp* - acceleration scheme
- PyKokkos† - acceleration scheme

Method	L-2 Norm
Pure Python (CPU)	1.74×10^{-4}
Numba (Threading)	2.06×10^{-4}
Numba (PyOmp)	1.70×10^{-4}
PyKokkos (OpenMP)	1.46×10^{-4}

Table 1: L-2 errors of solutions produced kernels as compared to data from MC/DC

- PyKokkos-Base† - builds Kokkos libraries to be linked to
- Matplotlib - plotting and output
- PyYaml - input file handling
- PyTest - testing

† Package onerous/impossible depending on platform to install, related features have been disabled for tests. *Package is in direct conflict with another, related features have been disabled.

4 Results

4.1 Test Problem Results

A test problem with a reference solution for a homogeneous slab with vacuum bounds on either side (L=1 cm, $\Delta x = 0.01\,\text{cm}$, $\Sigma_a = \Sigma_f = \Sigma_s = 0.333\,\text{cm}^{-1}$) at 1×10^5 particles was used to confirm that kernels where producing the correct solution. Figure 3 shows the scalar fluxes of all methods as compared to a reference solution. The figure shows that all methods lie on top of each other and virtually (as far as MC solutions at the given the particle count are concerned) match. Table 1 shows a clearer picture with L-2 error between MC/DC-TNT solutions and the reference solution being suitable for this number of particles.

4.2 Runtime Results

Test case 1 at 1×10^8 particles provides a substantial enough problem to require acceleration. Table 2 shows the decrease in run time of all methods considered thus far. The single thread pure Python implementation took almost 18 hours to complete with any acceleration method bringing that down to a matter of mere minuets. This demonstrates the power of any method to increase Python based performance by at least two orders of magnitude on CPU based machines. PyKokkos presents superior performance than both Numba implementations. However where PyKokkos fails compared to Numba is in the compilation time of kernels. Kernels are "warmed up" at the beginning of a simulation to enable the computation to be done all at once. Table 3 shows how Pykokkos takes almost 7 times as long to compile, though both finish in a matter of mere seconds.

4.3 Ease of Use

We also consider the difficulty of these implementations when coming from a pure Python code. The Numba CPU implementation was easy for simpler kernels, functioning as advertised with a single compiler flag. More complex kernels, especially ones designed to be run in parallel, had to

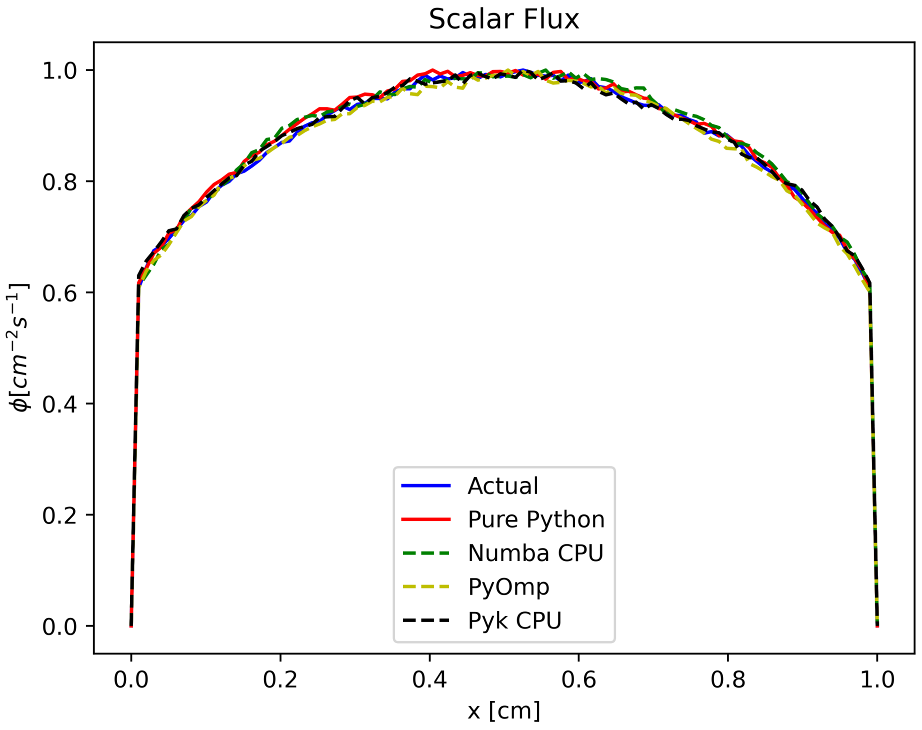


Figure 3: Scalar flux produced by test problem as compared to a reference solution.

Method	Advance	Integration
Pure Python (CPU)	5.140×10^4	5.297×10^4
Numba (Threading)	1.887×10^2	2.323×10^2
Numba (PyOmp)	2.876×10^2	3.825×10^2
PyKokkos (OpenMP)	1.480×10^2	1.548×10^2

Table 2: Run time of `Advance` kernel and integration test in seconds, using 1×10^8 particles and 16 threads (if method is parallelized); kernel compilation time not included.

Method	Compilation time
Numba (Threading)	4.99
Numba (PyOmp)	5.66
PyKokkos (OpenMP)	37.50
PyKokkos (CUDA)	39.72

Table 3: First run translation and compilation wall-clock times for all kernels, in seconds.

be reformed and use in kernel numba commands to work. Also, we found that when Numba wasn't able to produce kernels for any reason it's error messages directing us to the issues where either cryptic or nonexistent.

We found working within Pykokkos to be more difficult than Numba for the CPU implementation. Fueled again by cryptic runtime errors, frustrating type casting issues, and a lack of documentation both for the build process as well as module commands. We expect as PyKokkos matures these issues will be alleviated. It should also be stated the allure of PyKokkos is not only a slight increase of performance on CPU implementations when compared to Numba but also in its portability, with promises of CUDA GPU implementations with changing a single variable. Early implementations of PyKokkos CUDA on MC/DC-TNT does seem to confirm this.

5 Conclusions and Future Work

Significant work remains before we can select a method to implement in MC/DC. Our immediate goals are to complete the PyKokkos CUDA and Numba CUDA implementations before finally moving onto the Mako templating engine method. We will also implement the AZURV1 [3] transient benchmark to further validate MC/DC-TNT as well as examine performance under different nuclear material data regimes.

6 Acknowledgments

This work was supported by the Center for Exascale Monte-Carlo Neutron Transport (CEMeNT) a PSAAP-III project funded by the Department of Energy, grant number: DE-NA003967.

References

[1] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. A performance portability framework for python. In *Proceedings of the International Conference on Supercomputing*, pages 467–478. Association for Computing Machinery, 6 2021.

[2] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[3] B.D. Ganapol, R.S. Baker, J.A. Dahl, and Raymond E. Alcouffe. Homogeneous Infinite Media Time-Dependent Analytical Benchmarks, 2001.

[4] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 9 2020.

[5] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[6] E. E. Lewis and W. F. Miller. *Computational Methods of Neutron Transport.* Wiley, New York, NY, 1984.

[7] Timothy G. Mattson, Todd A. Anderson, Giorgis Georgakoudis, Konrad Hinsen, and Anshu Dubey. PyOMP: Multithreaded Parallel Programming in Python. *Computing in Science and Engineering*, 23(6):77–80, 11 2021.

[8] Ryan G McClarren. *Computational Nuclear Engineering and Radiological Science Using Python*. Academic Press, 2018.

[9] Jackson P. Morgan. Mcdc-tnt: Monte carlo dynamic code - transient neutronics testbed, March 2022.

[10] Lena Oden. Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing. In *Proceedings - 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2020*, pages 216–223, 2020.